from Interactive Theorem Proving to Integrated Theorem Proving

Makarius Wenzel

November 2016
Abstract

Interactive theorem proving was historically tied to the read-eval-print loop, with sequential and synchronous evaluation of prover commands given on the command-line. This user-interface technology was adequate when Robin Milner introduced his LCF proof assistant in the 1970s, but today it severely restricts the potential of multicore hardware and advanced IDE front-ends.

The Isabelle Prover IDE breaks this loop and retrofits the read-eval-print phases into an asynchronous model of document-oriented proof processing. Instead of feeding a sequence of commands into the prover process, the primary interface works via edits over immutable document versions. Execution is implicit and managed by the prover in a timeless and stateless manner, making adequate use of parallel hardware.

PIDE document content consists of the theory sources (with dependencies via theory imports), and auxiliary source files of arbitrary user-defined format: this allows to integrate other languages than Isabelle/Isar into the IDE. A notable application is the Isabelle/ML IDE, which can be also applied to the system itself, to support interactive bootstrapping of the Isabelle/Pure implementation.

Further tool integration works via "asynchronous print functions" that operate on already checked theory sources. Thus long-running or potentially non-terminating processes may provide spontaneous feedback while the user is editing. Applications range from traditional proof state output (which often consumes substantial run-time) to automated provers and dis-provers that report on existing proof document content (e.g. Sledgehammer, Nitpick, Quickcheck in Isabelle/HOL). It is also possible to integrate "query operations" via additional GUI panels with separate input and output (e.g. for manual Sledgehammer invocation or find-theorems).

Thus the Prover IDE orchestrates a suite of tools that help the user to write proofs. In particular, the classic distinction of ATP and ITP is overcome in this emerging paradigm of Integrated Theorem Proving.
History
TTY loop ($\approx 1979$)

- user drives prover, via manual copy-paste
- inherently synchronous and sequential
Proof General and clones (≈ 1999)

- user drives prover, via **automated copy-paste and undo**
- inherently **synchronous and sequential**
PIDE: Prover IDE (∼ 2009)

**Approach:**

**Prover** supports asynchronous document model natively

**Editor** continuously sends source edits and receives markup reports

**Tools** may participate in document processing and markup

**User** constructs document content — assisted by

  GUI rendering of cumulative PIDE markup
**PIDE: Prover IDE (≈ 2009)**

**Approach:**

*Prover* supports asynchronous document model natively

*Editor* continuously sends source edits and receives markup reports

*Tools* may participate in document processing and markup

*User* constructs document content — assisted by
  - GUI rendering of cumulative PIDE markup

**Challenge:** introducing genuine interaction into ITP

- many conceptual problems
- many technical problems
- many social problems
Isabelle/jEdit Prover IDE (2016)

---
advanced user interaction
Automatically tried tools (2016)

→ advanced tool integration
Isabelle/PIDE building blocks

**jEdit**: sophisticated text editor implemented in Java
http://www.jedit.org

**Scala**: higher-order functional-object-oriented programming on JVM
http://www.scala-lang.org

**PIDE**: general framework for Prover IDEs based on Scala with parallel and asynchronous document processing

**Isabelle/jEdit**:  
- main example application of the PIDE framework  
- default user-interface for Isabelle  
- filthy rich client: 2 cores + 4 GB RAM minimum
PIDE architecture
The connectivity problem

Design principles:

- **private** protocol for prover connectivity
  (asynchronous interaction, parallel evaluation)
- **public** Scala API
  (timeless, stateless, static typing)
PIDE protocol functions

- `type protocol_command = name → input → unit`
- `type protocol_message = name → output → unit`
- outermost state of protocol handlers on each side (pure values)
- asynchronous streaming in each direction

→ editor and prover as stream-processing functions
Approximative rendering of document snapshots

1. editor knows text $T$, markup $M$, and edits $\Delta T$ (produced by user)
2. apply edits: $T' = T + \Delta T$ (immediately in editor)
3. formal processing of $T'$: $\Delta M$ after time $\Delta t$ (eventually in prover)
4. temporary approximation (immediately in editor):
   $\hat{M} = \text{revert } \Delta T; \text{retrieve } M; \text{convert } \Delta T$
5. convergence after time $\Delta t$ (eventually in editor):
   $M' = M + \Delta M$
Document content and execution
Prover command transactions

- “small” toplevel state $st$: $Toplevel.state$
- command transaction $tr$ as partial function over $st$
  
  we write $st_0 \xrightarrow{tr} st_1$ for $st_1 = tr st_0$
- general structure: $tr = read; eval; print$

Interaction view:

$$tr\ st_0 =$$

- \textbf{let} $eval = read ()$ \textbf{in} $st_1$ \textbf{—} read does not require $st_0$
- \textbf{let} $st_1 = eval st_0$ \textbf{in} $st_1$ \textbf{—} main transition $st_0 \rightarrow st_1$
- \textbf{let} $() = print st_1$ \textbf{in} $st_1$ \textbf{—} print does not change $st_1$

Important: purely functional transactions with managed output
Command scheduling

Sequential R-E-P Loop:

\[ st_0 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_1 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_2 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_3 \cdots \]
Command scheduling

Sequential R-E-P Loop:

\[ st_0 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_1 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_2 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_3 \cdots \]

PIDE 2011/2012:

\[ \downarrow \xrightarrow{\text{read}} \quad \downarrow \xrightarrow{\text{read}} \quad \downarrow \xrightarrow{\text{read}} \quad \cdots \]

\[ st_0 \xrightarrow{\text{eval}} st_1 \xrightarrow{\text{eval}} st_2 \xrightarrow{\text{eval}} st_3 \cdots \]

\[ \downarrow \xrightarrow{\text{print}} \quad \downarrow \xrightarrow{\text{print}} \quad \downarrow \xrightarrow{\text{print}} \quad \cdots \]
Command scheduling

**Sequential R-E-P Loop:**

\[ st_0 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_1 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_2 \xrightarrow{\text{read}} \xrightarrow{\text{eval}} \xrightarrow{\text{print}} st_3 \xrightarrow{\vdots} \]

**PIDE 2011/2012:**

\[ \downarrow \text{read} \quad \downarrow \text{read} \quad \downarrow \text{read} \quad \vdots \]

\[ st_0 \xrightarrow{\text{eval}} st_1 \xrightarrow{\text{eval}} st_2 \xrightarrow{\text{eval}} st_3 \xrightarrow{\vdots} \]

\[ \downarrow \text{print} \quad \downarrow \text{print} \quad \downarrow \text{print} \]

**PIDE 2013/2014:**

\[ \downarrow \text{read} \quad \downarrow \text{read} \quad \downarrow \text{read} \quad \vdots \]

\[ st_0 \xrightarrow{\text{eval}} st_1 \xrightarrow{\text{eval}} st_2 \xrightarrow{\text{eval}} st_3 \xrightarrow{\vdots} \]

\[ \downarrow \downarrow \text{forks} \quad \downarrow \downarrow \text{prints} \quad \downarrow \downarrow \text{forks} \quad \downarrow \downarrow \text{prints} \]

Document content and execution 15
Global structure: directed acyclic graph (DAG) of theories

Local structure:

entries: linear sequence of command spans, with static command_id and dynamic exec_id

perspective: visible and required commands, according to structural dependencies

overlays: print functions over commands (with arguments)
Document nodes

**Global structure:** directed acyclic graph (DAG) of theories

**Local structure:**

- **entries:** linear sequence of command spans, with static `command_id` and dynamic `exec_id`
- **perspective:** visible and required commands, according to structural dependencies
- **overlays:** print functions over commands (with arguments)

**Notes:**

- for each document version, the command exec assignment identifies results of (single) `eval st` or (multiple) `print st`
- the same execs may coincide for different versions
- non-visible / non-required commands remain unassigned
Document edits

**Key operation:** update ↦ assignment

**datatype** edit = Dependencies | Entries | Perspective | Overlays

**val** Document.update: version_id → version_id → (node × edit) list → state → (command_id × exec_id list) list × state

**Notes:**

• document update restructures **hypothetical execution**
• **command exec assignment** is acknowledged quickly
• actual **execution is scheduled** separately
  → protocol thread remains reactive with reasonable latency
Execution management

Prerequisites:
- native threads in Poly/ML (D. Matthews, 2006 . . . )
- future values in Isabelle/ML (M. Wenzel, 2008 . . . )

Execution in PIDE 2013/2014:

Hypothetical execution: lazy execution outline with symbolic assignment of exec_ids to eval and prints

Execution frontiers: conflict avoidance of consecutive versions
  \[ \text{Execution.start: } \text{unit} \rightarrow \text{execution_id} \]
  \[ \text{Execution.discontinue: } \text{unit} \rightarrow \text{unit} \]
  \[ \text{Execution.running: } \text{execution_id} \rightarrow \text{exec_id} \rightarrow \text{bool} \]

Execution forks: managed future groups within execution context
  \[ \text{Execution.fork: } \text{exec_id} \rightarrow (\text{unit} \rightarrow \alpha) \rightarrow \alpha \text{ future} \]
  \[ \text{Execution.cancel: } \text{exec_id} \rightarrow \text{unit} \]

Document content and execution 18
PIDE backend implementation
Bidirectional byte-channel:
- pure byte streams with block-buffering
- high throughput
- TCP socket; not stdin/stdout

Message chunks:
- explicit length indication
- block-oriented I/O

Text encoding and character positions:
- reconcile ASCII, ISO-Latin-1, UTF-8, UTF-16
- unify Unix / Windows line-endings
- occasional readjustment of positions
YXML transfer syntax:
- markup trees over plain text
- simple and robust transfer syntax
- easy upgrade of text-based application

XML/ML data representation
- canonical encoding / decoding of ML-like datatypes
- combinator library for each participating language, e.g. SML:
  
  ```
  type α Encode.T = α → XML.tree list
  Encode.string: string Encode.T
  Encode.list: α Encode.T → α list Encode.T
  ```

- **untyped** data representation of typed data
- **typed** conversion functions
**Problem:** round-trip through several sophisticated syntax layers

**Solution:** execution trace with **markup reports**

**Example:** semantic markup for **domain-specific formal languages**
Conclusions
Achievements

Renovation and reform . . .
of Interactive/Integrated Theorem Proving
for new generations of users

Paradigm shift . . .
adequate use of multicores hardware with pervasive parallelism

Document-oriented approach . . .
for user interaction and tool integration

→ Towards *The Archive of Formal Proofs* as one big document!
Lessons learned

- Substantial reforms of LCF-style theorem proving are possible, with big impact on infrastructure, but little impact on existing tools.

- Parallel processing is relatively easy, compared to the difficulties of asynchronous user interaction and tool integration.

- Real-world frameworks like JVM/Swing impose technical side-conditions and challenges, notably for multi-platform support.
Try it yourself!

**Current release:** February 2016
http://isabelle.in.tum.de

**Next release:** December 2016
http://isabelle.in.tum.de/website-Isabelle2016-1-RC2